VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Predikátová logika 1. řádu a Hornovy klauzule
First Order Predicate logic and Horn Clauses

2014 Bc. Martin Blahovský

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Blahovský**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Predikátová logika 1. řádu a Hornovy klauzule
First Order Predicate Logic and Horn Clauses

Zásady pro vypracování:

Cílem této práce je aplikace řešící problematiku převodu formulí predikátové logiky 1. řádu (PL1) na Hornovy klausule. Jako nedílná součást je teorie související s touto problematikou.
V práci:
1. Rozeberte teorii predikátové logiky 1. řádu (jazyk, sémantika).
2. Popište dokazování platnosti úsudků či tautologií v predikátové logice 1. řádu (obecná rezoluční metoda).
3. Popište Hornovy klausule a jejich omezení, popište syntax programovacího jazyka Prolog a strategii řízení výpočtu.
4. Vytvořte případovou studii pro převod formulí predikátové logiky 1. řádu na Hornovy klausule v syntaxi programovacího jazyku Prolog (implementace).

Seznam doporučené odborné literatury:

[1] Duží M.: Matematická logika, VŠB-TUO
[2] Švejdar V.: Logika, neúplnost, složitost a nutnost, ACADEMIA 2003
[3] Sochor A.: Klasická matematická logika, KAROLINUM 2001
[4] Polák J.: Prolog, GRADA 1992

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Marek Menšík, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení:

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 5.5.2014

.....

Podpis

Rád bych poděkoval vedoucímu své diplomové práce Mgr. Markovi Menšíkovi, Ph.D. za trpělivost a ochotu při konzultacích a za všechen volný čas, který mi věnoval.

Abstrakt:

Diplomová práce popisuje základní charakteristiku predikátové logiky 1. řádu a řeší problematiku převodu formulí z PL1 na Hornovy klausule. Je definována abeceda a gramatika predikátové logiky. Práce interpretuje formule a jazyk a shrnuje dokazovací metody. Především je kladen důraz na ověřování platnosti úsudků pomocí obecné rezoluční metody. Dále je popsána syntaxe programovacího jazyka Prolog a strategie řízení výpočtů. Součástí diplomové práce je aplikace pro převod formulí z predikátové logiky 1. řádu na Hornovy klausule v syntaxi programovacího jazyka Prolog.

Klíčová slova:

Predikátová logika 1. řádu, PL1, rezoluční metoda, Hornovy klausule, Prolog, stromová struktura formule.

Abstract:

This Thesis describes basic characteristic of the first order predicate logic and solves issue of formulas transfer from PL1 to Horn clauses. It defines an alphabet and grammar of predicate logic. The Thesis interprets formulas and language and summarizes deductive methods. Primarily, the emphasis is placed on verifying the argument validity by using the general resolution method. It also describes the syntax of the programming language Prolog and the strategy of management derivation. The Thesis includes an application for formulas transfer from first order predicate logic to horn clauses in syntax of the programming language Prolog.

Key words:

First order predicate logic, PL1, resolution method, Horn clause, Prolog, tree structure of the formula.

OBSAH:

1. Úvod

Cílem mé diplomové práce je popsat teorii týkající se predikátové logiky 1. řádu a Hornových klauzulí. Práce je rozdělena na 3 hlavní kapitoly.

Úvodní kapitola je věnována jazyku predikátové logiky, je zde popsána abeceda a gramatika PL1. Následně je popsána interpretace jazyka a interpretace formulí PL1 a jejich převod z přirozeného jazyka do jazyka predikátové logiky 1. řádu. Součástí kapitoly je i popis dokazovacích metod v PL1 s hlavním zaměřením na dokazování platnosti úsudků pomocí obecné rezoluční metody.

Druhá kapitola je věnována logickému programování. Logické programování se využívá například ve vědě, expertních systémech, zdravotnictví, ale především v umělé inteligenci. Důraz v této diplomové práci je kladen na programovací jazyk Prolog. Je zde popsána syntaxe a sémantika jazyka Prolog, rysy jazyka a strategie řízení výpočtu. Dále jsou zde popsány Hornovy klauzule, které jsou pro programovací jazyk Prolog zásadní, jelikož na nich je programování v Prologu založeno.

Poslední kapitola je věnována aplikaci pro převod formulí predikátové logiky prvního řádu na Hornovy klausule, které jsou zapsány v syntaxi programovacího jazyka Prolog.

2. Predikátová logika

V kapitole týkající se predikátové logiky 1. řádu jsem čerpal z těchto pramenů: [1], [2], [3], [4].

V matematice a logice se pod pojmem predikátová logika označuje formální odvozovací systém používaný převážně k popisu matematických teorií a vět.

Predikátová logika je rozšířením výrokové logiky. Výroková logika nedokáže vyjádřit některá složitější tvrzení o matematických strukturách. Zatímco výroková logika se zabývá jednoduchými a deklarativními výroky, predikátová logika prvního řádu zavádí jako nadstavbu predikáty a kvantifikátory. Predikátová logika 1. řádu si všímá struktury vět. V každé větě rozlišuje individua, o nichž, se něco predikuje. Predikát intuitivně chápeme jako vlastnost nebo vztah a nabývá booleovské hodnoty (*pravda* - *nepravda*). Individuum je prvek z nějaké množiny (univerza) a predikát je relace na této množině.

Predikátová logika 1. řádu (PL1) umožňuje analyzovat elementární výroky do úrovně vlastností daných objektů, tzv. individuí (prvky univerza diskurzu) a jejich vztahů. Mějme zadaná individua „Sportovec“ a „Člověk“. Všechny prvky univerza diskursu(individua) lze dále kvantifikovat, tedy např. „Všichni lidé jsou sportovci“ nebo „Existují lidé, kteří jsou sportovci“. Pomocí těchto vlastností můžeme sestavovat výroky, o kterých rozhodujeme zda jsou pravdivé či nepravdivé a můžeme z nich odvozovat závěry.

Kromě predikátové logiky 1. řádu existují i logiky vyšších řádů. Predikátové logiky vyšších řádů se odlišují tím, že umožňují analyzovat výroky do úrovně vlastnosti vlastností, vlastnosti funkcí, apod.

V této teoretické kapitole je pozornost věnována úvodu do celé logiky s primárním zaměřením na predikátovou logiku 1. řádu včetně metod pro dokazování platnosti úsudků či tautologií v PL1.

2.1. Predikátová logika prvního řádu

Predikátová logika 1. řádu je formální systém používaný především v matematice, filozofii a informatice.

Jazyk predikátové logiky 1. řádu poskytuje prostředky pro zkoumání úsudků, které nejsou elementární. PL1 umožňuje sledovat vlastnosti předmětů a vztahy mezi předměty z pevně dané oblasti (univerza diskurzu). PL1 si všímá struktury vět.

Rozlišuje v každé větě individuum, resp. individua, o němž, resp. o nichž, se něco predikuje, tedy se zkoumají jeho vlastnosti, či vztahy. Predikát intuitivně chápeme jako vlastnost nebo vztah. Většina z toho, co jsme formulovali ve výrokové logice zůstává v platnosti i v rámci predikátové logiky.

Predikátová logika 1. řádu zachycuje logickou strukturu jednotlivých výroků. Výrok je tvrzení, o němž má smysl prohlásit, zda je pravdivé či nepravdivé.

V PL1 lze odvodit platnost jednotlivých výroků, které ve výrokové logice nejsou platné, viz. následující příklad:

Mějme následující výroky (+ označení výrokovými symboly):

Každý člověk je smrtelný. (p)

Sokrates je člověk. (q)

Sokrates je smrtelný. (r)

Ve výrokové logice by tento úsudek odpovídal tvrzení $\{ p , q \models r \}$, což odpovídá formuli: $(p \wedge q) \supset r$. Tyto uvedené 3 výroky jsou z hlediska výrokové logiky elementární a nezávislé navzájem. Ve skutečnosti mají ovšem tyto výroky vnitřní komponenty, které jsou strukturované a které mezi jednotlivými výroky zprostředkovávají vazby.

Termín „člověk“ se vyskytuje ve výrocích p i q. Termín „smrtelný“ se vyskytuje ve výrocích p i r. Termín „Sokrates“ se vyskytuje ve výrocích q i r.

Formule $(p \wedge q) \supset r$ není tautologie. Tedy ani úsudek $p , q \models r$ není platný, i když úsudek demonstrovaný příkladem evidentně platný je.

V predikátové logice zapíšeme zadaný úsudek touto formulí:

$$\forall x [P(x) \supset Q(x)], \; P(s) \models Q(s)$$

kde,

- $\forall x$ je značení pro všeobecný kvantifikátor.
- P, Q jsou dané vlastnosti entit z universa diskursu. V tomto konkrétním případě interpretujeme jako vlastnosti „být smrtelný“ a „být člověkem“.
- x je předmětová (individuová) proměnná, která patří do dané předmětné oblasti, tzv. universa diskursu.
- s je předmětná (individuová) konstanta z dané předmětné oblasti. V tomto konkrétním příkladu je to Sokrates.

Následující podkapitoly budou věnovány syntaxi a sémantice PL1.

2.1.1. Abeceda predikátové logiky 1. řádu

Jazyk predikátové logiky prvního řádu je na rozdíl od přirozených jazyků, jako např. čeština, zcela formální. Můžeme tedy mechanicky určit, zda je daný výraz utvořen správně, dle zadaných pravidel. Rozlišujeme základní dva druhy výrazů. Jsou jimi termy a formule. Termy intuitivně představují objekty. Formule zahrnují predikáty a mohou být pravdivé nebo nepravdivé.

Termy a formule jsou v predikátové logice symbolizovány jako řetězce symbolů. Jednotlivé symboly tvoří abecedu jazyka.

Symboly abecedy dělíme především na logické a na mimo-logické. Logické symboly mají vždy stejný význam. Např. logický symbol konjunkce „$\wedge$“ má vždy pouze jeden význam („a“, „a zároveň“). Oproti tomu význam mimo-logických symbolů se liší podle dané interpretace. Např. tedy predikátový symbol P ve formuli $P(x)$ můžeme interpretovat tak, že x je filosof, x je sportovec, apod.

Abeceda predikátové logiky se skládá z následujících částí:

a) Logických symbolů:

   i. Spočetné množiny individuových proměnných Var = $\{x,y,z, \ldots, x_1,x_{2,\ldots}\}$

   ii. Symbolů pro logické spojky: $\neg$ (negace), $\wedge$ (konjunkce), $\vee$ (disjunkce), $\supset$ (implikace), $\equiv$ (ekvivalence)

   iii. Symbol pro všeobecný kvantifikátor $\forall$ a symbol pro existenční kvantifikátor $\exists$

b) Speciálních (mimologických) symbolů:

   i. Množina predikátových symbolů $P = \{P,Q,R, \ldots\}$
   ii. Množina konstantních symbolů $C = \{a,b,c, \ldots\}$
   iii. Množina funkčních symbolů $F = \{f,g,h, \ldots\}$

c) Pomocných symbolů: závorky a čárka.

Pro každý *predikátový* i *funkční* symbol máme dáno přirozené číslo n, které je větší nebo rovno 1. Tomuto číslu n říkáme arita nebo také četnost *predikátového* symbolu nebo *funkčního* symbolu. Tyto symboly tedy mohou být unární, binární, ternární atd. Obecně je nazýváme

n-árními *predikátovými* nebo *funkčními* symboly. Arita udává počet individuových proměnných, které jsou argumenty *predikátového* nebo *funkčního* symbolu.

2.1.2. Gramatika predikátové logiky 1. řádu

Skladbu neboli syntax jazyka tvoří gramatika. Gramatika je tvořena ze dvou prvků – abecedy a gramatických pravidel(pravidel syntaxe).

Abecedu jazyka tvoří konečná množina symbolů – znaků abecedy. Abeceda je množina symbolů, kterými v jazyce disponujeme a které používáme.

Gramatiku jazyka tvoří soustava gramatických pravidel, díky kterým je umožněno z prvků abecedy jazyka konstruovat zřetězení symbolů jazyka, která jsou správně utvořenými formulemi.

Gramatická pravidla jsou definována tzv. induktivní formou od jednoduchého ke složitějšímu. To znamená, že z jednoduchých formulí skládáme složitější a z nich pak dále skládáme další formule.

Gramatická pravidla tvorby:

a) Termy:

Každý symbol proměnné *x, y,* … je term. Termy jsou ukazatele na prvky univerza diskursu.

Jsou-li $t_1,\ldots,t_n$ ($n \geq 0$) termy a je-li f *n*-ární funkční symbol, pak výraz $f(t_1,\ldots,t_n)$ je term; pro $n = 0$ se jedná o nulární funkční symbol, neboli individuovou konstantu (značíme *a, b, c,* …).

Pouze výrazy, které splňují výše uvedené vlastnosti jsou termy.

b) Atomické formule:

Je-li *P* *n*-ární predikátový symbol a jsou-li $t_1,\ldots,t_n$ termy, pak výraz $P(t_1,\ldots,t_n)$ je atomická formule.

Jsou-li t_1 a t_2 termy, pak výraz $(t_1 = t_2)$ je atomická formule.

c) Molekulární formule:

Každá atomická formule je formule.

Je-li výraz A formule, pak $\neg A$ je formule.

Jsou-li výrazy A a B formule, pak výrazy $(A \vee B)$, $(A \wedge B)$, $(A \supset B)$, $(A \equiv B)$ jsou formule.

Je-li x proměnná a A formule, pak výrazy $\forall xA$ a $\exists xA$ jsou formule.

Pouze výrazy, které splňují výše uvedené vlastnosti jsou formule.

2.1.3. Shrnutí predikátové logiky 1. řádu

Základní vlastností v predikátové logice 1. řádu je, že jediným povoleným typem proměnné jsou individuové proměnné. Pouze individuové proměnné lze vázat kvantifikátory.

Zápis výsledných formulí můžeme zjednodušit na základě dodržení konvencí o vynechávání závorek:

Elementární formule a formule nejvyššího řádu není třeba závorkovat, jsou vynechány vnější závorky. Elementární formule je formule řádu 0, která neobsahuje žádné funktory, tedy ani kvantifikátory. Formule vyšších řádů obsahují funktory, tedy kvantifikátory nebo logické spojky.

Po dodržení priority funktorů ve formuli není třeba psát závorky. Nejvyšší prioritu mají kvantifikátory $(\forall, \exists)$, poté v daném pořadí mají prioritu tyto logické spojky: $\neg, \wedge, \vee, \supset, \equiv$.

Za předpokladu, že o prioritě vyhodnocení formule nerozhodnou ani závorky ani priorita, tak vyhodnocujeme formuli zleva doprava.

Jelikož jsou logické spojky konjunkce a disjunkce asociativní, není třeba zapisovat závorky při zápisu většího počtu konjunkcí a disjunkcí.

2.2. Sémantika predikátové logiky 1. řádu

2.2.1. Interpretace formulí

Sémantika, neboli význam formulí predikátové logiky 1. řádu, je dána jejich interpretací. Zajímá-li nás, zda daná formule PL1 je pravdivá, či ne, pak je taková otázka nesmyslná, pokud nevíme, jak je formule interpretována a co znamená.

Interpretace formule spočívá nejdříve ve stanovení předmětné oblasti, tedy jaký je obor proměnnosti (individuových) proměnných. Zvolíme si tzv. universum diskursu, což je neprázdná množina, jejíž prvky nazýváme entity či individua.

Predikátové symboly vyjadřují vztahy mezi individui. Každému *n*-árnímu predikátovému symbolu přiřadíme jistou *n*-ární relaci, tedy podmnožinu kartézského součinu nad daným universem. Pokud se jedná o unární predikátový symbol, tedy $n = 1$, pak přiřadíme danou podmnožinu universa. Funkční symboly vyjadřují *n*-ární funkce nad universem.

Nyní po provedení výše uvedených kroků máme formuli interpretovanou. Můžeme tedy vyhodnotit její pravdivost či nepravdivost v této interpretaci.

Dalším krokem je práce s proměnnými. Proměnné v predikátové logice prvního řádu dělíme na volné a vázané.

Těmto proměnným přiřazujeme individua, tedy prvky z universa. Přiřazení prvku z universa proměnné se nazývá *valuace*, neboli ohodnocení. Máme-li formuli, která obsahuje nějaké volné proměnné, tak vyhodnocení pravdivosti či nepravdivosti v dané interpretaci můžeme provést pouze v závislosti na valuaci (ohodnocení) volných proměnných. Při některé valuaci může být formule pravdivá v dané interpretaci, v jiné *valuaci* může být nepravdivá.

Definice volné a vázané proměnné:

Výskyt proměnné x ve formuli A je vázaný, jestliže je součástí nějaké podformule $\forall x B(x)$, nebo $\exists x B(x)$ formule A.

Proměnná x je vázaná ve formuli A, má-li v A vázaný výskyt. Výskyt proměnné x ve formuli A, který není vázaný, nazýváme volný. Jestliže tedy existuje ve formuli A proměnná x, která není vázána kvantifikátorem, je volná.

Proměnná x je volná ve formuli A, má-li v A volný výskyt.

Formule, v níž každá proměnná má buď všechny výskyty volné, nebo všechny výskyty vázané, se nazývá formulí s čistými proměnnými.

Formule, která neobsahuje žádnou volnou proměnnou se nazývá uzavřená. Formule, která obsahuje alespoň jednu volnou proměnnou se nazývá otevřená. Nechť $x_1, x_2, \ldots, x_n$ jsou všechny volné proměnné formule A. Potom uzavřenou formuli $\forall A =_{df} \forall x_1 \forall x_2 \ldots \forall x_n A$ resp. $\exists A =_{df} \exists x_1 \exists x_2 \ldots \exists x_n A$, nazýváme generálním resp. existenčním uzávěrem formule A.

Formuli, která vznikne z formule A korektní substitucí termu t za proměnnou x označujeme symbolem $A(x/t)$. Korektní substituce musí splňovat tyto dvě pravidla:

Při substituci je nutné nahradit všechny volné výskyty proměnné x ve formuli A. Substituovat je možno pouze volné výskyty proměnné x ve formuli A.

Žádná individuová proměnná, která vystupuje v termu t se pro provedení substituce x/t nesmí stát vázanou proměnnou ve formuli A. V takovém případě by byl term t za proměnnou x ve formuli A nesubstituovatelný.

Symbolem $A(x_1,x_2,...,x_n / t_1,t_2,...,t_n)$ označujeme formuli, která vznikne z formule A korektními substitucemi x_i/t_i *pro* $i = 1, 2,...,n$. Všechny formule tvaru $A(x_1, x_2,...,x_n / t_1, t_2,...,t_n)$ nazýváme instancemi formule A.

Definice ohodnocení :

Ohodnocení (valuace) individuových proměnných je zobrazení e, které každé proměnné x přiřazuje hodnotu $e(x) \in M$ (prvek univerza).

Ohodnocení termů e^* indukované ohodnocením proměnných e je induktivně definováno takto:

$e^*(x) = e(x)$

$e^*(f\ (t_1,\ t_2,...,t_n)) = f_M\ (e^*(t_1),\ e^*(t_2),...,e^*(t_n))$, kde f_M je funkce přiřazená v dané interpretaci funkčnímu symbolu f.

2.2.2. Interpretace jazyka

Interpretace jazyka predikátové logiky prvního řádu, nebo také interpretační struktura je tvořena těmito třemi objekty:

a) Universem diskursu, což je neprázdná množina M, jejíž prvky jsou individua.

b) Interpretaci funkčních symbolů jazyka, která přiřazuje každému n-árnímu funkčnímu symbolu f určité zobrazení f_M: $M^n \rightarrow M$.

c) Interpretaci predikátových symbolů jazyka, která přiřazuje každému n-árnímu predikátovému symbolu p jistou n-ární relaci p_M nad M, tj podmnožinu Kartézského součinu M^n.

Definice splnitelnosti a pravdivosti formulí:

Formule A je splnitelná v interpretaci I, jestliže existuje ohodnocení e proměnných takové, že platí $|=_I A\ [e]$.

Formule A je pravdivá v interpretaci I, značíme $\models_I A$, jestliže pro všechna možná ohodnocení e individuových proměnných platí, že $\models_I A[e]$.

Formule A je splnitelná, jestliže existuje interpretace I, ve které je splněna, tj. jestliže existuje interpretace I a valuace e takové, že $\models_I A[e]$. Taková interpretace I a valuace e, tedy dvojice $\langle I,e \rangle$, pro kterou platí $\models_I A[e]$, se nazývá model formule.

Formule A je tautologií (logicky pravdivá), značíme $\models A$, jestliže je pravdivá v každé interpretaci.

Formule A je kontradikcí, jestliže nemá model, tedy neexistuje interpretace I, která by formuli A splňovala.

2.3. Převod z přirozeného jazyka do jazyka PL1

Chceme-li převést větu z přirozeného jazyka do jazyka predikátové logiky prvního řádu, tak ji musíme analyzovat podle pravidel PL1. Mezi tato pravidla patří:

Volba predikátových a funkčních konstant je libovolná, ovšem nesmí dojít ke kolizi vlastností, funkcí nebo vztahů.

Pro výrazy jako např. „všechny“, „všichni“, „každý“, „nikdo“, atd. je v PL1 použit všeobecný kvantifikátor $\forall$.

Pro výrazy jako např. „někdo“, „něco“, „někteří“, atd. je v PL1 použit existenční kvantifikátor $\exists$.

Stejně jako ve výrokové logice používáme k převodu do jazyka PL1 symboly logických spojek pro negaci, konjunkci, disjunkci, implikaci a ekvivalenci.

Pro označení predikátů používáme velká písmena. Výsledkem převodu je formule, která popisuje strukturu a logickou formu věty.

Ukázka převodu do jazyka PL1:

1) Někteří studenti jsou nadaní => $\exists x\,[S(x) \wedge N(x)]$
2) Všichni studenti jsou líní => $\forall x\,[S(x) \supset L(x)]$

2.4. Rozhodnutelnost

Rozhodnutelnost je pojem z oblasti logiky. Vyjadřuje, zda existuje konečný algoritmus, který pro libovolně zadanou formuli určí, zda je v dané teorii dokazatelná nebo není. Pokud pro

danou teorii existuje konečný algoritmus, tak je nazývána rozhodnutelnou. V opačném případě se jedná o nerozhodnutelnou.

Formule je tedy rozhodnutelná pouze v případě, že existuje algoritmus, který dokáže vždy rozhodnout, zda zadaná formule je tautologie, kontradikce nebo splnitelná. V případě, že takový algoritmus neexistuje, tak je formule nerozhodnutelná.

Na rozdíl od výrokové logiky je problém logické pravdivosti v predikátové logice nerozhodnutelný. To znamená, že neexistuje všeobecný algoritmus, který by naprosto vždy s jistotou rozhodl, zda zadaná formule je tautologie, kontradikce nebo splnitelná.

2.5. Dokazovací metody

Predikátová logika má dva typy dokazování platnosti jednotlivých úsudků. Dokazovací metody jsou buď sémantické nebo syntaktické.

Sémantické metody se zabývají především sémantikou dané formule, neboli jejím modelem. V predikátové logice nás tedy zajímá interpretace dané formule.

Syntaktické metody se naopak zabývají pouze syntaxí, neboli tvarem, ale nezajímají se o význam formule.

Mezi sémantické metody patří:

a) Metody zabývající se modely
b) Vennovy diagramy

Mezi syntaktické metody patří například:

a) Sémantická tabla
b) Rezoluční metoda

2.5.1. Obecná rezoluční metoda

Rezoluční metoda v predikátové logice je zobecněním rezoluční metody výrokové logiky. Jak již bylo uvedeno výše, jedná se o syntaktickou dokazovací metodu.

Pomocí rezoluční metody ověřujeme platnost úsudků a tautologičnost formule. Oproti výrokové logice je rezoluční metoda v predikátové logice prvního řádu složitější díky bohatší struktuře formulí.

Rezoluční metoda v PL1 se dá aplikovat pouze na formule ve speciální konjunktivní formě. Jedná se o tzv. Skolemovu klauzulární formu.

Obecná rezoluční metoda se stala základem pro programovací jazyk Prolog a pro logické programování.

Ověřování platnosti úsudků v PL1:

a) Přímá metoda:

   Premisy musí mít pouze všeobecné kvantifikátory. Výsledek vyplývá z rodičovských klauzulí. Formuli převedeme do Skolemovy klauzulární formy. Postupně je aplikováno rezoluční pravidlo na dané klauzule. Tímto máme generovány rezolventy. Výsledná rezolventa logicky vyplývá z rodičovských klauzulí.

b) Nepřímá metoda:

   V prvním kroku je třeba znegovat závěr. Tento negovaný závěr připojíme k množině předpokladů. Formuli převedeme do Skolemovy klauzulární formy. Postupně aplikujeme na klauzule rezoluční pravidla a snažíme se dospět k prázdné klauzuli.

   Dospějeme-li k prázdné klauzuli, došlo ke sporu, což pro původní nenegovaný závěr znamená, že úsudek je platný.

Ověřování tautologičnosti formule:

Provádí se nepřímo. Znegujeme formuli a převedeme ji do Skolemovy klauzulární formy. Poté na klauzule aplikujeme rezoluční pravidla a snažíme se dospět k prázdné klauzuli, tedy ke sporu.

Pokud dojdeme k prázdné klauzuli. Tak jsme zjistili, že původní nenegovaná formule je tautologie.

2.5.2. Skolemizace

V této kapitole bude popsán algoritmus převodu formule do Skolemovy klauzulární formy.

Krok 1:

Utvoření existenčního uzávěru formule A. (Zachovává splnitelnost.)

Krok 2:

Eliminace nadbytečných kvantifikátorů. (Ekvivalentní krok.) Z formule A vypustíme všechny kvantifikátory $\forall x_i$, $\exists x_i$, v jejichž rozsahu se nevyskytuje proměnná x_i.

Krok 3:

Přejmenování proměnných. (Ekvivalentní krok) Přejmenujeme všechny proměnné, které jsou v A kvantifikovány více než jednou tak, aby všechny kvantifikátory měly navzájem různé proměnné.

Krok 4:

Eliminace spojek $\supset$, $\equiv$ podle těchto vztahů (Ekvivalentní krok.): $(A \supset B) \Leftrightarrow (\neg A \vee B)$, $(A \equiv B) \Leftrightarrow (\neg A \vee B) \wedge (\neg B \vee A)$

Krok 5:

Přesun spojek $\neg$ dovnitř. (Ekvivalentní krok.)

Krok 6:

Přesun kvantifikátorů doprava (Ekvivalentní krok). Provádíme náhrady podle těchto ekvivalencí (Q je kvantifikátor $\forall$ nebo $\exists$; © je symbol $\wedge$ nebo $\vee$; A, B neobsahují volnou proměnnou x):

$Qx\,(A \,©\, B(x)) \Leftrightarrow A \,©\, Qx\,B(x)$, $Qx\,(A(x) \,©\, B) \Leftrightarrow Qx\,A(x) \,©\, B$

Krok 7:

Eliminace existenčních kvantifikátorů (zachovává splnitelnost). Provádíme postupně Skolemizaci podformulí $Qx\,B(x)$, $Qx\,A(x)$, které jsme obdrželi v předchozím kroku 6, tedy náhradu existenčně kvantifikovaných formulí formulemi bez existenčního kvantifikátoru.

Krok 8:

Přesun všeobecných kvantifikátorů doleva (Ekvivalentní krok, neboť jsme již provedli krok 3. a platí ekvivalence dle 6).

Krok 9:

Použití distributivních zákonů. (Ekvivalentní krok) Provedeme postupné náhrady vlevo formulemi vpravo: $(A \wedge B) \vee C \Leftrightarrow (A \vee C) \wedge (B \vee C)$, $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$

3. Logické programování

V kapitole týkající se logického programování jsem čerpal z těchto pramenů: [1], [5], [6], [7], [8] , [9].

Logické programování má základ v predikátové logice prvního řádu. Celý koncept logického programování vychází z toho, že máme zadáno tzv. universum diskursu a znalosti. Fakta i pravidla jsou v něm definována pomocí formulí predikátové logiky prvního řádu, respektive Hornových klauzulí.

Výpočet v logických programech vychází ze zjednodušování daných formulí pomocí pravidel a zjišťování důsledků, které z těchto formulí vyplývají.. Právě tímto se logické paradigma programování liší od imperativního. Výsledek je totiž obsažen v samotném zadání logického programu. Programátor neurčuje způsob, jakým se bude výpočet provádět, ale pouze zadává data programu ve formě, která je pro program vyhovující. Logický program je posloupnost příkazů(procedur), podmíněných(pravidel) i nepodmíněných(faktů). Cílová klauzule zadává dotazy, na které program nalezne odpověď.

Výpočet logického programu vychází z principu rezoluce predikátové logiky, a proto musí tomuto faktu odpovídat i forma zadaných dat.

Logické programování nám dává možnost poměrně jednoduchým a intuitivním způsobem popsat nějakou situaci, ve které hledáme odpovědi na otázky. Zda je něco možné a případně za jakých podmínek, což výrazně usnadňuje programátorovi práci, protože se nemusí zabývat algoritmy, které by tento problém vyřešily. Logický program je totiž deklarativní. Je specifikováno, „co se má provést“, ale programátor neurčuje, „jak se to má provést“.

Logické programování nabízí 2 způsoby vyhodnocování dílčích výsledků. Prvním je klasický bihodnotový systém, kdy odpověď může nabývat pouze hodnot 1 a 0, tedy *true* a *false*, neexistuje nic mezi tím. Druhým způsobem je tzv. *fuzzy logika*, které je věnována následující kapitola.

3.1.1. Ukázka důkazu

Zadání:

Každý, kdo má rád Jiřího, bude spolupracovat s Milanem.

Milan nekamarádí s nikým, kdo kamarádí s Láďou.

Petr bude spolupracovat pouze s kamarády Karla.

Jestliže Karel kamarádí s Láďou, pak Petr nemá rád Jiřího.

Formalizace:

$\forall x\ [R(x, J) \supset S(x, M)]$

$\forall x\ [K(x, L) \supset \neg K(M, x)]$

$\forall x\ [S(P, x) \supset K(x, Kr)$

———————————

$K(Kr, L) \supset \neg R(P, J)$

Převod do Skolemovy klauzulární formy a aplikace rezolučních pravidel na klauzule:

Klauzule:

1.	$\neg R(x, J) \vee S(x, M)$	1. předpoklad
2.	$\neg K(y, L) \vee \neg K(M, y)$	2. předpoklad
3.	$\neg S(P, z) \vee K(z, Kr)$	3. předpoklad
4.	$K(Kr, L)$	negovaný závěr
5.	$R(P, J)$	negovaný závěr
6.	$\neg K(M, Kr)$	rezoluce 4., 2., substituce *y*/Kr
7.	$\neg S(P, M)$	rezoluce 3., 6., substituce *z*/M
8.	$\neg R(P, J)$	rezoluce 1., 7., substituce *x*/P
9.	#	rezoluce 5., 8.

Došli jsme k prázdné klauzuli. Množina s negovaným závěrem je sporná. Tudíž původní nenegovaný závěr vyplývá z předpokladů a tento úsudek je platný, což znamená, že závěr vyplývá ze zadaných premis.

3.2. Fuzzy logika

Pro účely vyhodnocování na jiném principu, než je bihodnotový systém, je zde tzv. fuzzy logika. Fuzzy logika slouží, na rozdíl od dvoustavové logiky, k definování stavů, které nemají tak ostře vymezené hranice. Jedná se např. o pojmy: menší, větší, studený, teplý, vlažný, apod. Pohybujeme se na intervalu <0,1>. Prvek do dané množiny patří více nebo méně na základě vágnosti.

Klasická logika je podmnožinou fuzzy logiky, která je rozšířena o práci s hodnotami pohybujícími se mezi *true* a *false*, tudíž s částečnou pravdou. Jedná se tedy o vícehodnotovou logiku, jejímž cílem je vypracovat model fenoménu vágnosti (příslušnosti). Vágnost určuje jak

moc se daná proměnná přibližuje k pravdivé hodnotě *true*. Nabývá všech hodnot z intervalu < 0, 1 >, kterých může být nekonečně mnoho.

Cílem fuzzy logiky je vypracovat tzv. model vágnosti (příslušnosti). Vágnost určí, jak moc se proměnná přibližuje k pravdivé hodnotě *true*. Fuzzy logika může být vhodnější pro řadu rozhodovacích úloh než klasická bihodnotová logika, protože usnadňuje návrh složitých řídících systémů.

Hlavní využití fuzzy logiky:

- Rozpoznávání obrazu – strojové vidění, zpracování obrazu
- Robotika
- ABS, řízení motoru a další podsystémy vozidel(Subaru, Honda).
- Řízení výtahů(Mitsubishi)
- Spotřební elektronika – pračky, myčky, apod.
- DSS – systémy na podporu rozhodování

Klasickým příkladem pro využití fuzzy logiky může být popisování řízení auta, kde by mohly být zadány tyto 2 pravidla:

"Pokud na semaforu svítí jen oranžové světlo **a zároveň** vzdálenost od křižovatky je malá **a zároveň** rychlost auta je malá **pak** sešlápni brzdu střední silou."

"**Pokud** na semaforu svítí jen oranžové světlo **a zároveň** vzdálenost od křižovatky je velmi malá **a zároveň** rychlost auta je střední **pak** sešlápni plyn velkou silou."

Tato pravidla mohou být Fuzzy logikou jednoduše interpretována. Díky nim lze lehce modelovat a zejména automatizovat postupy z mnoha oblastí lidského života.

Nyní bude následovat konkrétní ukázka v jazyce Prolog. Bude definován vágní pojem *young*.

```
young :# fuzzy_predicate([(0,1),(35,1),(45,0),(120,0)]).
```

Tento konkrétní případ definuje predikáty:

```
young(X,1):- X .>=. 0, X .<=. 35.
young(X,M):- X .>=. 35, X .<=. 45, 10*M .=. 45-X.
```

```
young(X,0):- X .>=. 45, X .<=. 120.
```

První pravidlo určuje, že model vágnosti osoby do 35 let je 1. Ve druhém pravidle, které má interval věku osob od 35 let do 45 let, se model vágnosti počítá dle vzorce:

$10 * M = 45 - X$

Například ro osobu ve věku 37 let bude model vágnosti 0,8. Naopak pro osobu ve věku 44 let bude model vágnosti 0,1.

Poslední pravidlo určuje, že model vágnosti osoby od 45 let je 0.

3.3. Hornovy klauzule

Při rezolučním odvozování můžeme pracovat s množinami libovolných klauzulí. Cena, kterou se platí za tuto obecnost je ve výpočtové složitosti. Délka důkazu sporu může být v některých případech exponenciální funkcí velikosti množiny klauzulí.

Výkonné algoritmy dokazování vět se proto opírají o heuristiky založené na rozsáhlých znalostech problémové oblasti. Rezoluční metoda v plné obecnosti proto není vhodná jako základ programovacího jazyka. Jedna z možností, jak zjednodušit hledání sporu, spočívá v omezení tvaru klauzulí. Ukázalo se, že rezoluční strategie odvozování jsou mnohem efektivnější na množinách Hornových klauzulí.

V logice, konkrétně ve výrokové logice se pod pojmem Hornova klauzule označuje speciální druh klauzule (disjunkce literálů). Literál je libovolná atomická formule, buď pozitivní nebo negativní. Hornova klauzule obsahuje nejvýše jeden pozitivní literál, všechny ostatní musí být negované.

Matematický zápis Hornovy klauzule:

$\neg P \vee \neg Q \vee \ldots \vee \neg T \vee U$

Hornovu klauzuli tak lze obecně zapsat jako implikaci ve formě:

$(P \wedge Q \wedge \ldots \wedge T) \supset U$

V prologu se toto pravidlo zapisuje touto formou:

```
U:- P,Q,…,T.
```

Jako Hornova formule se označuje taková formule, která je v konjunktivní normální formě, a která se skládá z Hornových klauzulí. Jako duální Hornova klauzule se pak označuje klauzule, která obsahuje nejvýše jeden negativní literál, ostatní literály jsou tedy pozitivní.

Logický program je tvořen konečnou množinou Hornových klauzulí, které obsahují nanejvýš jeden pozitivní literál. Každý program obsahuje fakta, pravidla a otázky.

Pro pravidla platí:

Hornova klauzule obsahuje právě 1 pozitivní literál a alespoň 1 literál negativní.

Pravidla můžeme chápat jako mechanismus, který popisuje konkrétní část světa. Např. můžeme definovat, že všichni synové jsou muži.

Zápis v prologu: `muž(X):-syn(X).`

Pro fakta platí:

Hornova klauzule obsahuje právě 1 pozitivní literál a nemá žádný negativní literál. Fakta chápeme jako jevy, které jsou vždy splnitelné. Např. můžeme definovat, že Martin a Michal jsou bratři.

Zápis v prologu: `bratr(Martin,Michal).`

Pro otázky platí:

Hornova klauzule neobsahuje žádný pozitivní literál, ale obsahuje alespoň 1 negativní literál. Takto vytvořenou klauzuli chápeme jako dotaz. Např. se můžeme ptát, zda Martin a Michal jsou bratři. Na takto položený dotaz získáme odpověď *true/false*.

Zápis v prologu: `?- bratr(Martin,Michal).`

3.4. Programovací jazyk Prolog

Název Prolog vznikl z anglického originálu „Programming in logic“. Programovací jazyk Prolog vznikl na přelomu 60. a 70. let minulého století. Autory Prologu jsou Robert Kowalski z univerzity v Edinburgu a Alain Colmerauer a Phillipe Roussel, oba pochází z univerzity Aix-Marseille.

Od počátku byl Prolog využíván při zpracování přirozeného jazyka a pro symbolické výpočty v různých oblastech umělé inteligence. V průběhu doby se z jazyka Prolog stal univerzální programovací prostředek pro softwarovou realizaci celé řady úloh umělé inteligence. Používá se v databázových a expertních systémech, zpracování přirozeného jazyka, rozpoznávání obrazu, aplikace v učících se systémech a aplikace v úlohách robotiky i jako nástroj pro řešení úloh.

Prolog se od klasických imperativních programovacích jazyků (Java, C/C++, C#, …) liší deklarativním způsobem programování. Tento způsob se vyznačuje tím, že klade důraz na to „co se má počítat“, aniž by bylo podrobně specifikováno „jak se to má počítat“.

U deklarativního způsobu programování chybí příkazy pro řízení běhu programu i příkazy pro řízení toku dat, což umožňuje plně se soustředit na popis relací a vytvářet tak snadno prototypy programů. Pomocí tohoto jazyku lze formulovat libovolnou úlohu přímo v jazyce logiky 1. řádu. U deklarativního programovacího jazyka jako je Prolog je potřeba znát pouze nutná fakta, která program vyhodnotí sám.

V programovacím jazyce Prolog je potřebné znalosti formulovat pomocí tzv. Hornových klauzulí:

$$(P_1 \wedge P_2 \wedge, \ldots, \wedge Pn) \supset A$$

Zápis tohoto pravidla v Prologu je zobrazen níže:

```
A:- P1,P2,…,Pn.
```

Základem tohoto jazyka je práce s termy, což jsou objekty a klauzulemi, které se dělí na fakta a pravidla. Programovací jazyk Prolog používá lineární strategii generování rezolvent s tzv. navrácením(backtracking). Je možno zadat formuli i ve formě, kdy za implikací následuje logická spojka OR:

$$(P_1 \wedge P_2 \wedge, \ldots, \wedge Pn) \supset (A \vee B)$$

V tomto konkrétním případě dojde k rozdělení formule na 2 pravidla a výpis v Prologu vypadá takto:

```
A:- P1,P2,…,Pn.

B:- P1,P2,…,Pn.
```

3.5. Rysy jazyka Prolog

Program v prologu definuje konečný počet relací pomocí příkazů, které popisují jejich vlastnosti. Můžeme říct, že program deklaruje vše, co se může vypočítat. Prolog se tímto podobá specifikačním jazykům, které dovolují oddělit problém úplné a korektní specifikace o problému efektivnosti výpočtu vytvářeného programu. Hlavní výhodou tohoto způsobu je, že je snazší upravit konkrétní specifikaci než hotový program. Způsob výpočtu logického programu je dán procedurální interpretací jeho příkazů.

Prolog pracuje s relacemi, který zahrnuje operace používané v relačních databázových modelech a dovoluje definovat dotazovací jazyky pro specifické potřeby uživatele databáze. Je tedy vhodným prostředkem i pro databázové aplikace.

Podmíněné příkazy Prologu mají tvar implikací:

A platí, jsou-li splněny podmínky $P_1, P_2, \ldots, P_N$.

Podobný tvar implikačních pravidel se používá v mnoha systémech umělé inteligence, především v expertních systémech.

Mezi hlavní rysy programovacího jazyka Prolog patří tyto vlastnosti:

Uživatel jazyka Prolog pouze specifikuje čeho má být při řešení úlohy dosaženo aniž by musel rozepisovat postup, jak toho má být dosaženo.

Není tedy potřeba formulovat úlohu ve formě algoritmů s využíváním řídících struktur klasických programů.

Další výhodou je jednoduchost syntaxe jazyka Prolog spolu s možností formulovat relace v přirozeném jazyce.

3.6. Databáze Prologu

Programování v jazyce Prolog spočívá v deklarování faktů o objektech a relacích mezi objekty. Základem je definování pravidel o objektech a relacích platících mezi nimi a v zodpovídání dotazů.

V logickém programování jsou fakta nepodmíněné příkazy a pravidla jsou podmíněné příkazy. Jak pravidla, tak i fakta se ukládají do společné databáze. Prolog nerozlišuje mezi programem a daty.

Fakta by měla mít určitou vypovídací schopnost. Chceme-li tedy definovat databázi, která by byla vhodná k dalšímu využití, tak je třeba zavést kromě objektů i jejich vlastnosti, vztahy nebo chování. Příklady faktů jsou zobrazeny níže:

```
muz(daniel).

zena(mirka).
```

Díky takto zadaným faktům máme v databázi uloženo , že Daniel je muž a Mirka je žena.

```
manzele(daniel,mirka).

rodic(daniel,alena).
```

```
rodic(mirka,alena).
```

Takto napsanou deklarací zavedeme fakt, že Daniel a Mirka jsou manželé a zároveň jsou i rodiči Aleny. Při zadávání faktů je třeba dávat si pozor na zavedené pořadí parametrů. Počet parametrů faktů není nijak omezen. Jen je třeba si dát pozor, zda jsou informace, které ukládáme srozumitelné a zda se navzájem nepopírají. Počet parametrů faktu nazýváme v Prologu aritou.

3.7. Metody strategie procházení grafu

Algoritmy pro efektivní procházení a vyhledávání v grafu musí splňovat tyto následující vlastnosti:

- Každou hranou projdeme vždy maximálně jednou, resp. jednou tam a jednou zpět.
- Hranou se vracíme zpět pouze tehdy, nevede-li z vrcholu již další cesta.
- Hranou, která vede do navštíveného vrcholu se ihned vracíme.

Takto vytvořený algoritmus je konečný a korektní. Konečnost plyne z první vlastnosti. V každém kroku projdeme jednou hranou, každou hranu jen jednou a hran je konečně mnoho.
Korektnost algoritmu v tomto případě znamená, že algoritmus projde všechny vrcholy a hrany, které jsou dosažitelné z výchozího vrcholu.

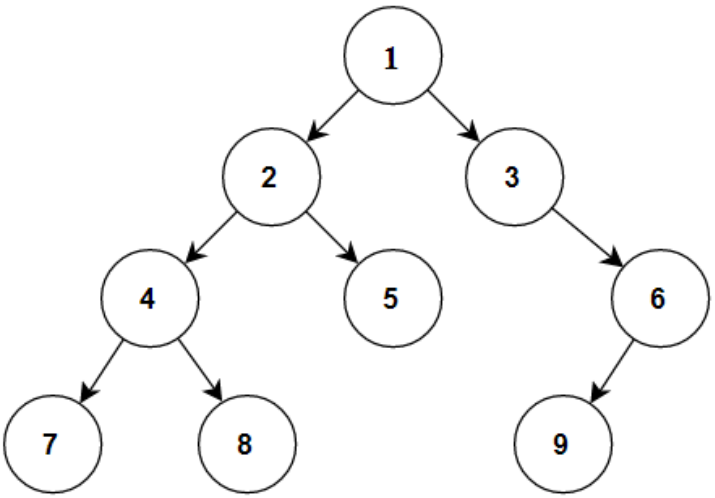
Procházení grafu se uskutečňuje podle toho, jak jsou hrany grafu uspořádány v databázi. Takové prohledávání nazýváme slepé.
Algoritmus prohledávání grafu, který není vybaven další informací, se nemá o co opřít a nezbývá mu než systematicky probírat všechny možnosti. Avšak i při slepém prohledávání existují dvě varianty pro prohledávání grafů. Jedná se o prohledávání grafu do hloubky a do šířky.

3.7.1. Prohledávání do šířky

Procházení do šířky (BFS z anglického Breadth first search). Jedná se o grafový algoritmus, který prochází postupně všechny vrcholy v daném grafu.

Algoritmus v prvním kroku projde všechny sousedy počátečního vrcholu, poté prochází sousedy sousedů atd. až projde všechny vrcholy v dané komponentě. Tento způsob procházení grafů se nazývá také „algoritmus vlny“, protože v každém kroku jsou nalezeny všechny uzly, které mají od počátečního vrcholu stejnou vzdálenost.

Tento postup zaručuje, že pokud cesta do cílového uzlu existuje, tak bude i nalezena, a to dokonce ta nejkratší. Tato varianta prohledávání grafu však může být z výpočetního hlediska velmi náročná, protože můžeme prozkoumávat mnoho odboček, které k cíli nevedou.



Obrázek 1: Prohledávání do šířky

Na obrázku máme zobrazeno, jak funguje algoritmus BFS. Začínáme prohledávání v kořenu stromu označeného „1“. Následně se prohledávají všechny uzly, které mají od kořene vzdálenost 1. Potom ty, které mají vzdálenost 2 atd.

Tento algoritmus se hodí například pro hledání nejkratší cesty v grafu mezi dvěma vrcholy. Hlavní výhodou algoritmu prohledávání do šířky je jistota, že cílový uzel bude skutečně nalezen. U prohledávání do šířky k navrácení nedochází. Každý uzel je tedy navštíven maximálně jednou.

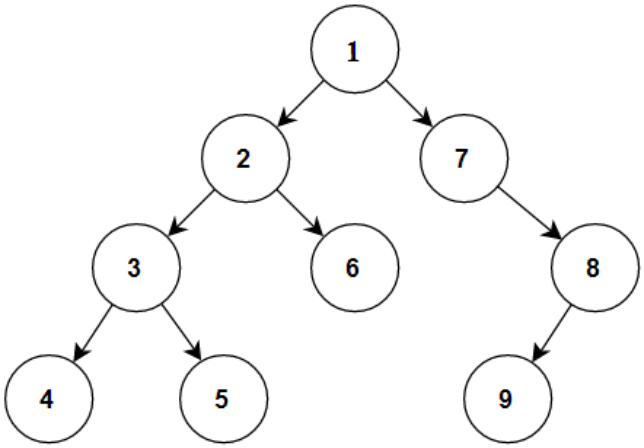
3.7.2. Prohledávání do hloubky

Procházení do hloubky (DFS z anglického Depth first search). Jedná se o algoritmus, který pro procházení grafů používá metodu backtrackingu. Tento algoritmus neprochází graf ve vlně jako BFS, ale prochází ho rekurzivně.

Algoritmus pracuje tak, že vždy pokračuje prvním následníkem každého vrcholu pokud jej ještě nenavštívil. Jakmile algoritmus narazí na vrchol, ze kterého již nelze dále pokračovat (buď neexistují další následníci nebo byli již všichni navštíveni), vrací se zpět backtrackingem.

Backtracking, neboli zpětné vyhledávání, je způsob řešení algoritmických problémů, který je založen na metodě pokus – omyl. Jedná se o vylepšené hledání řešení v tom, že velké množství

potencionálních řešení může být vyloučeno bez přímého vyzkoušení. Algoritmus backtrackingu je založen na procházení do hloubky možných řešení.



Obrázek 2: Prohledávání do hloubky

Na obrázku máme zobrazeno, jak funguje algoritmus DFS. V případě tohoto algoritmu postupujeme při prohlížení do hloubky po větvích. Nejprve je tedy prohlédnuta nejlevější větev, potom větve, které s ní mají nejdelší společnou větev. Po vyčerpání těchto možností se přechází k další větvi.

Při tomto způsobu postupujeme dopředu (přesněji do hloubky) rychleji, ale za tu cenu, že můžeme cíl minout nebo sledovat nekonečnou větev. Výhodou prohledávání do hloubky je nižší paměťová náročnost, protože se v paměti uchovávají pouze uzly na cestě od počátečního stavu ke stavu expandovanému. Hlavní nevýhodou tohoto algoritmu je, že nemáme zaručeno zda hledaný uzel doopravdy najdeme, jelikož algoritmus může procházet nekonečnou větev.

3.7.3. Ukázka výpočtu v Prologu

Je zadán program(množina klauzulí):

1. A
2. B
3. $C \vee \neg B$
4. $D \vee \neg A$
5. $D \vee \neg C$

Bude-li položen dotaz „Kdy platí D“, tedy „Kdy D vyplývá ze zadané množiny“, je třeba připojit 6. klauzuli $\neg D$ a provést rezoluci.

Jsou dvě možnosti, jak dokázat D z uvedených předpokladů 1-5:

a)

6. $\neg D$

7. $\neg A$ [6,4]

8. # [7,1]

b)

6. $\neg D$

7. $\neg C$ [6,5]

8. $\neg B$ [7,3]

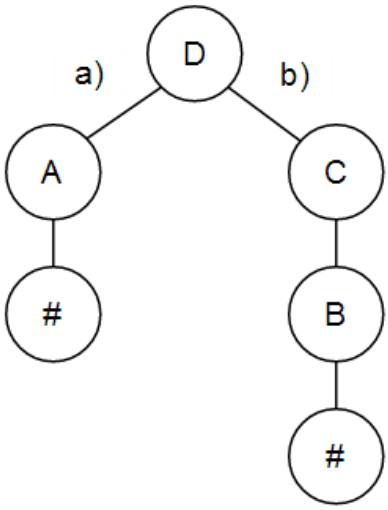
9. # [8,2]

V prologu zapíšeme zadané klauzule takto:

`A.`	fakt
`B.`	fakt
`C:-B.`	pravidlo(C pokud B)
`D:-A.`	pravidlo(D pokud A)
`D:-C.`	pravidlo(D pokud C)
`?-D.`	cíl(dotaz)

Při strategii prohledávání do hloubky je provedena nejdříve větev a), tedy zjistíme, že *D* platí za podmínky *A*, která je splněna. Až po splnění větve a) provedeme větev b). Zjistíme, že *D* platí za podmínky *C*, které platí za podmínky *B*, která je splněna.

Při strategii prohledávání do šířky dochází k paralelnímu procházení obou větví postupně. Nejdříve jsou tedy vygenerovány klausule 7 a 7', poté 8 a 8', apod. Celý postup je znázorněn ve výpočtovém stromu programu:



Obrázek 3: Výpočtový strom programu

Strategie prohledávání do hloubky prohledává postupně jednu větev za druhou. Tento algoritmus je efektivnější než prohledávání do šířky, avšak program může uvíznout v tautologii, tedy v nekonečné větvi.

Prolog volí pro řešení úlohy prohledávání stromu strategii prohledávání do hloubky. Toto řešení je sice výhodné z hlediska množství vznikajících nových cílových klauzulí, avšak výpočet se může vydat po nekonečné větvi.

Výhodou prohledávání do hloubky je, že vyžaduje, aby v paměti počítače byl uchováván pouze aktuálně řešený tvar dotazu a cesta, kterou byl tento dotaz odvozen z výchozího dotazu. Tyto informace jsou uloženy do zásobníku, který se nazývá rezoluční zásobník úlohy. Do zásobníku je v každém kroku výpočtu uložena tato trojice údajů: dotaz, navržená substituce a číslo klauzule programu použité spolu se substitucí k odvození dotazu. Klauzule jsou při vytváření programu vzestupně číslovány. Toto číslování je pro program neměnné.

Každý krok výpočtu programu hledá hodnoty proměnných, pro které jsou splněny všechny cílové klauzule

$?\text{-}C_1, \ldots, C_n.$

Je-li úspěšný, vydá odpověď (pokud jsou všechny cíle splněny) nebo generuje novou cílovou klauzuli (rezolventu), jejíž cíle budou řešeny v dalších krocích.

Řešení dotazu „$?\text{-}C_1,\ldots,C_n.$“ a vytváření rezolučního zásobníku úlohy probíhá takto:

1. Předpokládejme, že na začátku je zásobník prázdný.
2. Řešení aktuálního dotazu začíná prvním cílem zleva. Cíl C_1 se porovnává s hlavami příkazů s číslem vyšším než číslo poslední uvažované klauzule v pořadí, jak jsou v programu uvedeny. Hledá unifikace σ cíle C_1 s hlavou A příkazu. Rozlišují se 3 případy:

a) Unifikace existuje a A je hlava nepodmíněného příkazu. Provedením substituce σ pak vznikne pravdivý speciální případ A_σ formule A, který se shoduje se speciálním případem $C_{1\sigma}$ cíle C_1. Cíl C_1 je splněn pro hodnoty, které předepisuje substituce σ. Je-li $n > 1$, vznikne dosazením těchto hodnot do zbývajících cílů nová cílová klauzule

$?\text{-}C_{2\sigma},\ldots,C_{n\sigma}.$

jejíž cíle je třeba splnit. Je-li $n = 1$, výpočet končí uvedením hodnot proměnných, tj. vydáním odpovědi, která vznikne složením všech substitucí v zásobníku se substitucí σ. Je-li $n > 1$, pak je do zásobníku uložen cíl, unifikace σ a číslo použitého nepodmíněného příkazu. Aktuálním dotazem se stává „$?\text{-}C_{2\sigma},\ldots,C_{n\sigma}.$“, číslo poslední uvažované klauzule je 0 a algoritmus pokračuje od bodu 2.

b) Je-li A hlava podmíněného příkazu

$A\text{:-}P_1,\ldots,P_k.$

pak provedením substituce σ vznikne jeho speciální případ

$A_\sigma\text{:-}P_{1\sigma},\ldots,P_{k\sigma}.$

který podmiňuje splnění cíle $C_{1\sigma}$ shodného s hlavou A_σ splněním podmínek

$P_{1\sigma},\ldots,P_{k\sigma}$.

substituci σ provedeme ve všech cílech klauzule „`?-C1, …, Cn.`“ a cíl $C_{1\sigma}$ nahradíme podmínkami „`A:-P1,…,Pk.`“. Tím vznikne nová cílová klauzule

`?-P1σ,…,Pkσ,C2σ,…,Cnσ.`

do zásobníku je opět uložen cíl „`?-C1,…,Cn.`“, unifikace σ a číslo použitého podmíněného příkazu. Aktuálním dotazem se stává „`?-P1σ,…,Pkσ,C2σ,…,Cnσ.`“, číslo poslední uvažované klauzule je 0 a algoritmus pokračuje od bodu 2.

c) Není-li možné unifikovat cíl C_1 s hlavou žádného příkazu, nelze cíl C_1 splnit. Tento krok výpočtu je neúspěšný. Pak je třeba se vrátit na nižší úroveň a pokusit se o alternativní cestu. Ze zásobníku se vyjme poslední trojice: dotaz, substituce, číslo klauzule a algoritmus se vrací k bodu 2. Pokud je zásobník prázdný, byl již prohlédnut celý strom řešení a nebyl přitom nalezen prázdný cíl. Algoritmus končí neúspěchem.

3.8. Sémantika jazyka Prolog

Sémantika se týká významové stránky daného jazyka. V jazyce Prolog je sémantika relativně jednoduchá, jelikož operace disjunkce i konjunkce jsou komutativní operace. Takže nezáleží na pořadí, ve kterém jsou fakta definována. Prolog je zpracovává krok po kroku.

Prolog patří mezi tzv. deklarativní programovací jazyky, ve kterých uživatel popisuje pouze cíl výpočtu. Přesný postup, jakým se k výsledku program dostane, je pak ponechán na systému samotném. Děje se tak odvozením z faktů a pravidel uložených v databázi.

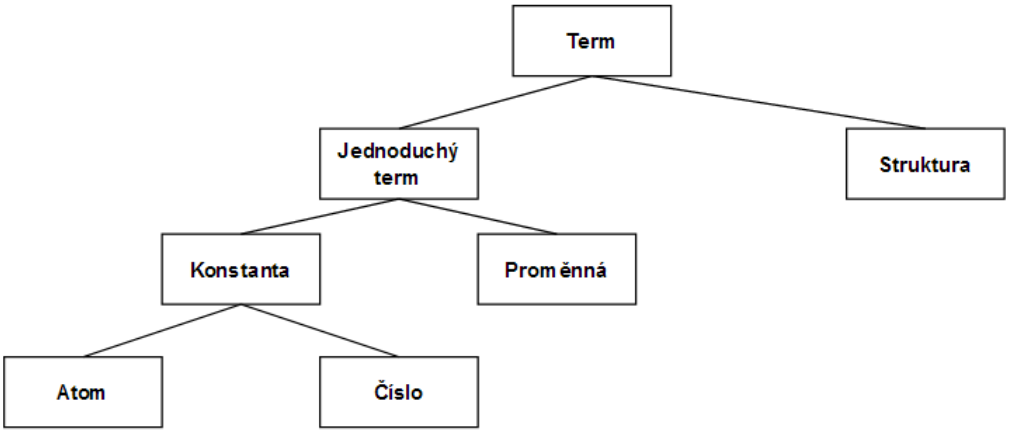
Deklarativní sémantika programu P definuje, jestli daný cíl je logickým důsledkem formulí, které tvoří program *P*, a pro jaké objekty toto platí. Z toho vyplývá, že pro „čistý Prolog“ deklarativní sémantika nezávisí na pořadí procedur a cílů v programu *P*.

3.9. Syntaxe jazyka Prolog

Programování v Prologu se skládá ze 3 základních kroků:

- Definice obecných pravidel, které platí mezi objekty a definice vztahů mezi nimi.
- Definice faktů, které jsou známy o objektech a vztahy mezi nimi.
- Formulace otázek o faktech a jejích vztazích.

Všechny objekty v prologu se nazývají termy. Termy jsou libovolné jevy, které můžeme popsat a zkoumat.



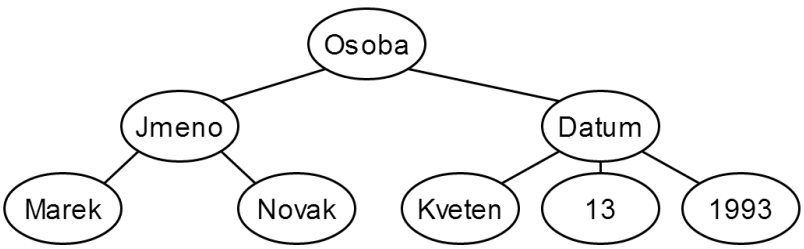
Obrázek 4: Datové typy Prologu

Termy tedy můžeme rozdělit na konstanty, proměnné a struktury.

- Struktura je určena funktorem a obsahuje posloupnost libovolného počtu termů, které jsou uzavřeny v závorkách. Každý funktor je určen svým názvem a aritou. Arita je počet termů.

  Příkladem struktury může být: student/2

```
osoba(jmeno('Marek', 'Novak'), datum(kveten,13,1993)).
```



Obrázek 5: Uložení struktury do stromové hierarchie

- Konstanty slouží k pojmenování objektů nebo relací. Konstantami mohou být:
  - Posloupnosti písmen, čísel a speciálních znaků, které začínají malým písmenem.
  - Celá čísla(mohou být i reálná).
  - Sekvence znaků uzavřených v apostrofech.

  Příkladem konstant mohou být:

```
777, auto, 'Prolog'
```

- Proměnné jsou vytvořeny posloupností znaků a číslic. Proměnná musí začínat velkým písmenem nebo znakem _ (podtržítko).

Příkladem proměnné může být:

`_lclvar, _` (volná proměnná) – není vázána na hodnotu tak jako ostatní proměnné.

Program v jazyce Prolog se skládá z posloupnosti klauzulí. Klauzule může být struktura, za níž je tečka. V tom případě se jedná o fakt, pokud neobsahuje proměnné, pak se jedná o tzv. základní fakt.

Nebo se může jednat o strukturu, která následuje znaky „ :- “ po nichž je uvedený libovolný počet struktur oddělených čárkami. Poslední struktura je ukončena tečkou. V tomto případě se jedná o pravidlo. Část vlevo od dvojznaku :- je hlava pravidla, napravo od dvojznaku je tělo pravidla.

Program může obsahovat dotazy, kterým říkáme cílové klauzule. Cílové klauzule v textu programu nesmějí obvykle obsahovat proměnné, na něž není vázána hodnota. Cílová klauzule zadává otázky, na které má program nalézt odpovídající odpovědi. Cestu, jak vyhodnotit zadané dotazy, najde překladač. Překladač tedy určí, co vyplývá ze zadané báze znalostí a jaké hodnoty je nutno substituovat unifikací za proměnné.

V prologu jsou zavedeny operátory, které slouží pro snazší zapisování a čtení termů. V následující tabulce jsou zobrazeny základní operátory, které v Prologu můžeme nalézt:

Operátor	Význam
:-	Definice pravidla
?-	Otázka
;	Logické nebo "|"
,	Logické a "&"
=	Porovnání
\=	Nerovná se
is	Vyčíslení
<, >, =<, >=	Porovnání
+, -, *, /, mod, div	Standartní matematické funkce
not	Negace

Tabulka 1: Syntaxe jazyka Prolog

Kromě těchto vestavěných operátorů si v Prologu můžeme vytvořit libovolné vlastní operátory. Nové operátory je třeba definovat následujícím způsobem:

```
:-op(priorita, specifikátor, jméno).
```

Priorita je číselná hodnota, která udává v jakém pořadí se mají operátory vyhodnocovat v případě, že výraz obsahuje operátorů více. Čím nižší číslo, tím vyšší priorita vyhodnocení daného operátoru. Pokud se vyskytne možnost, že výraz obsahuje operátory se stejnou prioritou, musí program vědět, jakým směrem má při výpočtu postupovat. Z tohoto důvodu se v operátoru definuje tzv. specifikátor.

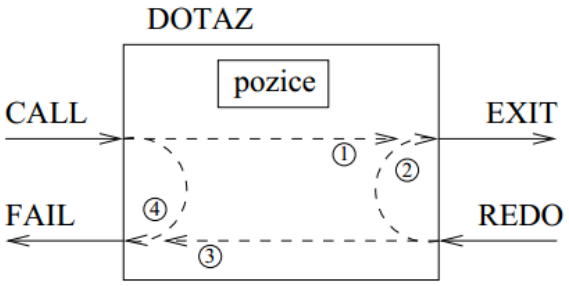
Specifikátory rozdělujeme podle směru v jakém se vyhodnocují a postavení vzhledem k operandu tímto způsobem:

- Infixový binární operátor:
  - *yfx* směr vyhodnocení operátoru zleva. Příkladem použití mohou být aritmetické operátory.
  - *xfy* směr vyhodnocení operátoru zprava. Příkladem je operátor *is*.
  - *xfx* zde nemá smysl hovořit o směru vyhodnocení. Nalevo i napravo mohou totiž být pouze operátory s vyšší vyhodnocovací prioritou. Příkladem použití může být porovnání.
- Prefixový unární operátor:
  - *fx* – operátor stojí před operandem. Příkladem použití je operátor not.
- Postfixový unární operátor:
  - *xf* – operátor stojí za operandem. Žádný standartní operátor tento specifikátor nevyužívá.

3.10. Grafický model výpočtu jazyka Prolog

Pro znázornění průběhu vyhodnocování cílů v jazyce Prolog a pro ladění programů se často používá tzv. blokový model výpočtu.

Každému volání predikátu odpovídá blok se 4 bránami. Dvě brány jsou vstupní a dvě jsou výstupní.

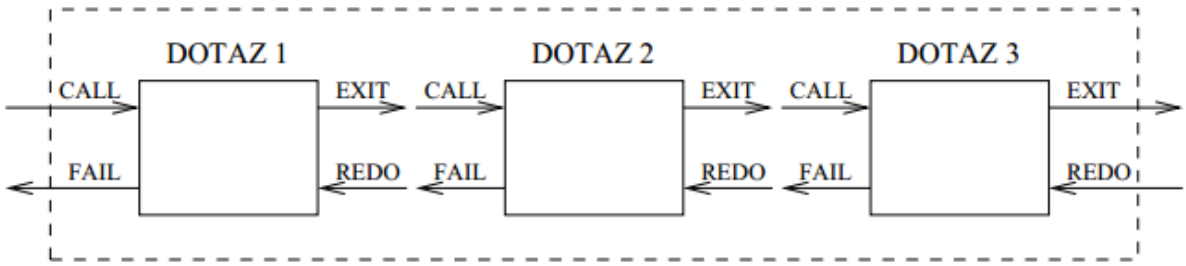


Obrázek 6: Grafický model výpočtu v prologu

Horní brány CALL a EXIT se používají při postupu výpočtu vpřed, zatímco spodní brány REDO a FAIL se používají pří navrácení výpočtu.

Vstupní brány označujeme CALL a REDO. CALL je základní vstup, kterým do bloku vstupujeme poprvé při požadavku na splnění cíle. Podaří-li se daný cíl splnit, tedy byla nalezena taková substituce, která převedla daný term na takový, který jde z programu odvodit, odchází výpočet branou EXIT k dalšímu cíli, který se bude plnit jako další.

Pokud dojde k tomu, že se následující cíl nepodaří splnit, tak se výpočet vrací pomocí brány FAIL z daného cíle do cíle předchozího do brány REDO. Tento návrat je signálem k tomu, že se má hledat další alternativní substituce pro splnění aktuálního cíle. Podaří-li se najít novou alternativní substituci, tak výpočet odchází branou EXIT opět k následujícímu bloku. Pokud se následující cíl opět nepodaří splnit, tak je výpočet opět navrácen branou FAIL k předcházejícímu bloku.



Obrázek 7: Blokové schéma složeného dotazu

Blokový model, který je vyobrazen výše znázorňuje složený dotaz. Vazba FAIL – REDO znázorňuje princip zpětného zřetězení. Složený dotaz se navenek chová jako jeden blok. Tímto způsobem můžeme dotazy do sebe libovolně skládat a vnořovat. Prakticky můžeme vytvořit

neomezenou hloubku vnoření. Dotaz ve svém důsledku může obsahovat i sám sebe a může tedy vytvářet i rekurzi.

Tímto grafickým modelem, který má uvedeny všechny vstupní i výstupní brány můžeme znázornit veškeré elementy prologovského programu. Blok grafického modelu může obsahovat klauzule, proceduru nebo i celý program.

Výstupní brána EXIT označuje úspěšnou, kladnou odpověď „*yes*“, případně s hodnotami proměnných (např. KDO = *prolog*). Naopak výstupní brána FAIL říká, že cíl nebyl splněn a vrátí uživateli odpověď „*no*“. V tomto případě říkáme, že došlo k neúspěchu při splňování cíle.

4. Případová studie

Hlavním cílem této diplomové práce je aplikace řešící problematiku převodu formulí predikátové logiky 1. řádu (PL1) na Hornovy klauzule v syntaxi programovacího jazyka Prolog. Problematika převodu formulí PL1 do požadovaného tvaru byla detailně teoreticky vysvětlena v předchozích kapitolách.

4.1. Implementace

Program je vytvořen jako webová aplikace. Aplikace je naprogramována v ASP.NET pomocí vývojového prostředí MS Visual Studio 2010. Logická vrstva aplikace je naprogramována v programovacím jazyce C#.

Pro testovací účely jsem vytvořil dočasný web na adrese hornclauses.aspone.cz, kde je program umístěn.

4.2. Vstupy aplikace

Vstupní data pro webovou aplikaci vychází z podkapitoly Abeceda predikátové logiky. Speciální značky pro logické spojky jsou nahrazeny textovými hodnotami. Funkční symboly použité v aplikaci jsou tyto.

Symboly pro logické spojky:

Negace ¬ je nahrazena znakem !.

Konjunkce ∧ je nahrazena znakem &.

Disjunkce ∨ je nahrazena znakem |.

Implikace ⊃ je nahrazena znakem >.

Symboly pro predikáty:

Predikáty začínají velkými písmeny např. *A,B, Student, Člověk*.

Symboly pro předmětové(individuové) proměnné:

Jsou psány malými písmeny např. *x, y, z*.

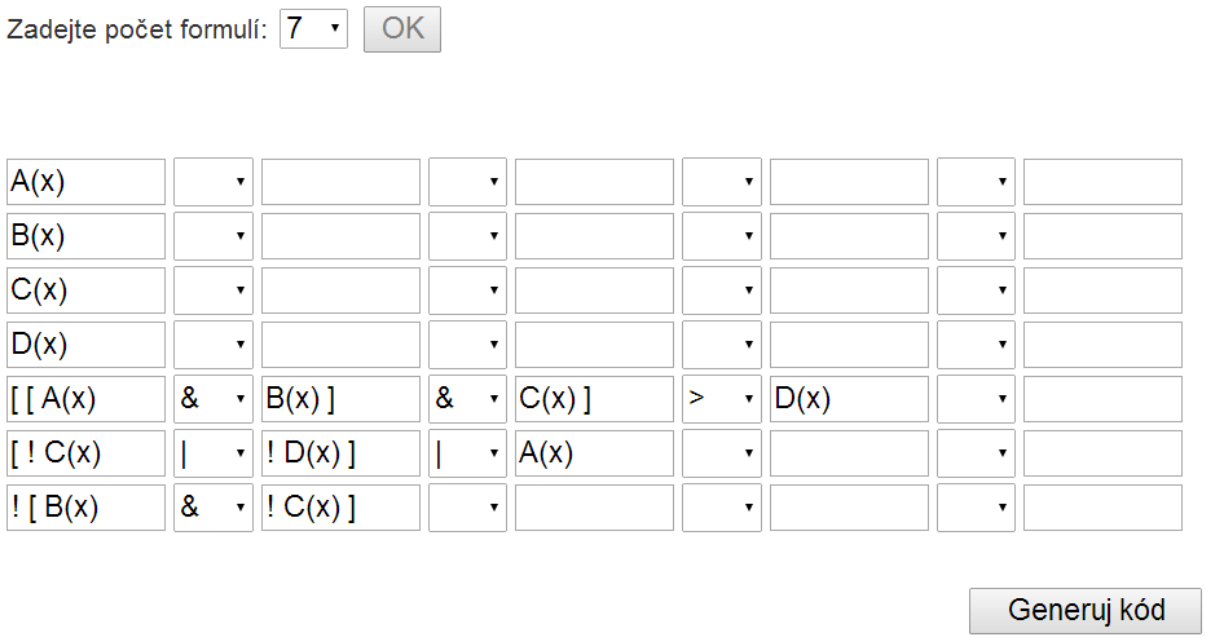
Pomocné symboly: (,), [,]

4.3. Uživatelské rozhraní

Hlavní snahou v programu bylo, aby zadávání formulí bylo intuitivní a jednoduché. V prvním kroku je uživatel vyzván, aby zadal počet premis se kterými bude program pracovat. Po vybrání počtu premis z DropDownListu a stisku tlačítka „*OK*“ budou jednotlivé komponenty vygenerovány.

K zadávání klauzulí je v programu připravena komponenta Predikat.ascx. Komponenta se skládá z Dropdownlistů, kterými vybíráme logické spojky a z Textboxů, do kterých jsou zadávány jednotlivé predikáty.

Ukázka zadání dat v aplikaci je zobrazena na následujícím obrázku:



Obrázek 8: Zadání vstupních dat aplikace

Důležitým pravidlem je správné uzávorkování formule. Každé dvě binární spojky musí být odděleny hranatými závorkami „[ ]“. Při zadávání dat v textboxech, kde jsou zadávány znaky pro negaci „!“ , prioritu operátorů „[ ]“ a samotné predikáty je nutno dodržovat pravidlo

odsazení jednotlivých znaků mezerami. Případné nadbytečné mezery, které by mohly být způsobeny chybou uživatele jsou odstraněny použitím regulárního výrazu.

Zadávání dat do komponent bylo vybráno z důvodu předcházení uživatelských vstupních chyb. Bylo specifikováno, že aplikace má být schopna pracovat s rovnicemi, které obsahují 4 logické spojky. V případě, že by uživatel potřeboval řešit složitější rovnice, tedy rovnice, které by obsahovaly 5 a více logických spojek je možno aplikaci jednoduše upravit. Řešením by bylo generování textboxů namísto předpřipravených komponent Predikat.ascx. V textboxu by uživatel nebyl limitován délkou rovnice. Samotná úprava v aplikaci pro tuto funkčnost by byla triviální.

4.4. Popis aplikace

Po zadání všech vstupních formulí a stisku tlačítka „Generuj Kód“ je zavolána metoda `PrintOutput(object sender, EventArgs e){}`. Tato metoda postupně prochází všechny zadané komponenty. Zde se metoda dělí na dva cykly. V prvním cyklu jsou vybrány komponenty, které obsahují pouze fakta. Ukázkou faktů mohou být například:

- *Student(Martin)*
- *A(x)*

Jakmile jsou zpracovány formule, které obsahují pouze fakta, tak je spuštěn cyklus, který prohledá vstupní komponenty a zpracuje postupně ty, které budou tvořit pravidla logického programu. Takovými formulemi mohou být například:

- $[\neg [A(x) \vee \neg B(x)] \wedge C(x)] \supset D(x)$
- $[\neg A(x) \vee \neg B(x)] \vee C(x)$

Nyní bude popsán algoritmus, jak program pracuje s jednotlivými komponentami(formulemi). Nejdříve je formule uložena do řetězce *sentence*. Tento řetězec je použit jako parametr pro metodu, která ze zadaného řetězce vytvoří binární strom. Jedná se o metodu `public static Node CreateInfixTree(string equation){}`.

Tato metoda vytváří instanci třídy Node.cs. Převod formule na stromovou strukturu je v programu jedna z nejdůležitějších funkcí. Stromová struktura totiž umožňuje mnohem jednodušší způsob provádění úprav nad zadanou formulí.

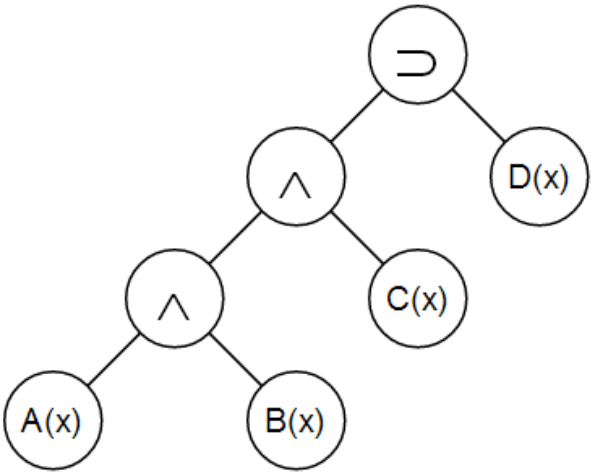
Procesem vytvoření stromové struktury analyzujeme aktuální formuli. Aktuální formule vždy obsahuje jednu z možností:

- Binární spojky – konjunkce, disjunkce, implikace.

- Unární spojka negace
- Predikát
- Závorky

Názorná ukázka, jak může vypadat stromová reprezentace zadané formule je zobrazena na obrázku níže.

Mějme zadanou formuli: $[\,[\,A(x) \wedge B(x)\,] \wedge C(x)\,] \supset D(x)$



Obrázek 9: Stromová reprezentace formule

Výsledná stromová struktura umožňuje jak pohyb od kořene k listům, tak i pohyb od jednotlivých listů směrem ke kořeni. Tato struktura je ideální pro provádění úprav na zadané formuli. Navrhnutá stromová struktura splňuje následující pravidla:

- Základním konstrukčním prvkem stromové struktury je uzel. Uzel je třída(Node.cs), která obsahuje:
    - Proměnnou „*Data*“ datového typu string, ve které je uložena hodnota uzlu.
    - Objekt „*Parent*“ typu uzel, ve které je uložen odkaz na rodičovský uzel.
    - Objekt „*Left*“ typu uzel, ve kterém je uložen odkaz na následující levý uzel.
    - Objekt „*Right*“ typu uzel, ve kterém je uložen odkaz na následující pravý uzel.

- Proměnnou „*FlagNegation*“ datového typu bool, což je příznak, který značí, zda je daný objekt negovaný.
- Výčtový typ „*Operation*“, který může nabývat hodnot {AND, OR, IMPL, NEG, NONE}. Konkrétní případ „NONE“ znamená, že se nejedná o logickou operaci, ale o predikát.

- Unární spojka negace má vždy levého následníka prázdného(*null*).
- Unární spojka negace má vždy pravého následovníka neprázdného.
- Binární spojky(konjunkce, disjunkce a implikace) mají vždy levého i pravého následovníka neprázdného.
- Kořen stromu nemá předchůdce. Objekt „*Parent*“ je *null*.
- Uzel, který má levého i pravého následníka prázdného(*null*), je listem.
- V listu může být pouze predikát.

Zadaná formule je tedy uložena ve stromové struktuře. Nyní je třeba upravit rovnici do požadovaného tvaru. K upravení stromu do požadovaného tvaru slouží rekurzivní metody `Browse a DestroyAND`.

Metoda `Browse` prochází stromem rekurzivně do hloubky. Podle priority formule vyhledá postupně logické spojky, kdy nejdříve je prohledávána levá větev stromu a v každém kroku rekurze rozdělí formuli na levou a pravou část. Takto je strom prohledán až k listům. Při zpětném čtení směrem od listů ke kořenu jsou prováděny v uzlech tyto ekvivalentní úpravy:

$A \supset B \equiv \neg A \vee B$

$A \wedge B \equiv \neg A \vee \neg B$

Po prohledání levé větve je stejný postup aplikován i na pravou větev stromu. Dále bude zobrazena metoda `Browse`.

```
public void Browse(bool neg = false)
{
    Browse(this, neg);
}
```

```
public void Browse(Node node, bool neg = false)
{
    switch (node.Operation)
    {
        case Operation.IMPL:
            {
                if (node.FlagNegation)
                {
                    node.Operation = Operation.AND;
                    node.Right.Browse(true);
                    node.Data = "&";
                }
                else
                {
                    node.Operation = Operation.OR;
                    node.Left.Browse(true);
                    node.Data = "|";
                }
                node.FlagNegation = !node.FlagNegation;
            }
            break;
        case Operation.AND:
            {
                node.Operation = Operation.OR;
                node.Left.Browse(true);
                node.Right.Browse(true);
                node.Data = "|";
            }
            break;
        case Operation.OR:
            {
                if (neg && node.FlagNegation)
                {
                    node.Left.Browse();
                    node.Right.Browse();
                }
                else
                {
                    node.Left.Browse(neg);
                    node.Right.Browse(neg);
                    if (neg)
                    {
                        node.Operation = Operation.AND;
                        node.Data = "&";
                    }
                }
            }
            break;
        case Operation.None:
            if (neg)
                node.FlagNegation = !node.FlagNegation;
            break;
    }
}
```

Po ukončení metody `Browse` je formule ve stromu zbavena implikací, ale v případě, že byla zadána složitější rovnice, tak se v uzlech stromu mohou ještě vyskytovat logické spojky $\wedge$. Z toho důvodu byla implementována rekurzivní metoda `DestroyAND`. Tato metoda opět prochází stromem nejdříve levou větví, kde provádí ekvivalentní úpravy až poté větví pravou.

```
public void DestroyAND(bool isRoot = false)
{
    DestroyAND(this, isRoot);
}

public void DestroyAND(Node node, bool isRoot)
{
    if (node.Operation == Operation.AND)
    {
        node.Operation = Operation.OR;
        node.Data = "|";

        node.Left.DestroyAND();
        node.Right.DestroyAND();
    }
    else if (node.Operation == Operation.OR)
    {
        if (!isRoot)
        {
            node.Operation = Operation.AND;
            node.Data = "&";
        }
        node.Left.DestroyAND();
        node.Right.DestroyAND();

        node.Operation = Operation.OR;
        node.Data = "|";

        node.Left.DestroyAND();
        node.Right.DestroyAND();
    }
    else if (node.Operation == Operation.None)
    {
        node.FlagNegation = !node.FlagNegation;
    }
}
```

Po provedení metody `DestroyAND` je již formule, která je uložena ve stromu zbavena, jak implikací, tak i konjunkcí a v uzlech stromu jsou pouze disjunkce. Nyní program pokračuje převodem formule ze stromové struktury na řetězec. K tomu slouží metoda `public override string ToString(){}`. Tato metoda opět rekurzivně prochází strom. Přečte jej a uloží do řetězce, který funkce navrátí.

Tento řetězec je následně předán jako vstupní parametr metodě `public static bool IsHornClause(string input){}`, která vyhodnotí, zda je zadaná formule Hornovou klauzulí nebo není. V případě, že metoda `IsHornClause` vrátí hodnotu *false*, tak zadaná formule není Hornovou klauzulí. V opačném případě, tedy když metoda vrátí *true,* tak je zavolána metoda

`public static string getPrologForm(string input){},` která přebírá jako vstupní parametr formulí, která byla celým algoritmem zpracována. Metoda `getPrologForm` vrátí řetězec, který je již v požadovaném tvaru Hornových klauzulí v syntaxi programovacího jazyka Prolog.

4.5. Výstupy aplikace

Výstupem aplikace je logický program v syntaxi deklarativního programovacího jazyka Prolog. Tento program se může skládat z faktů, pravidel a otázek.

Fakta obsahují právě jeden pozitivní literál a žádný negativní literál. Pravidla obsahují právě jeden pozitivní literál a alespoň jeden negativní literál. Otázky neobsahují žádný pozitivní literál, ale obsahují alespoň jeden negativní literál.

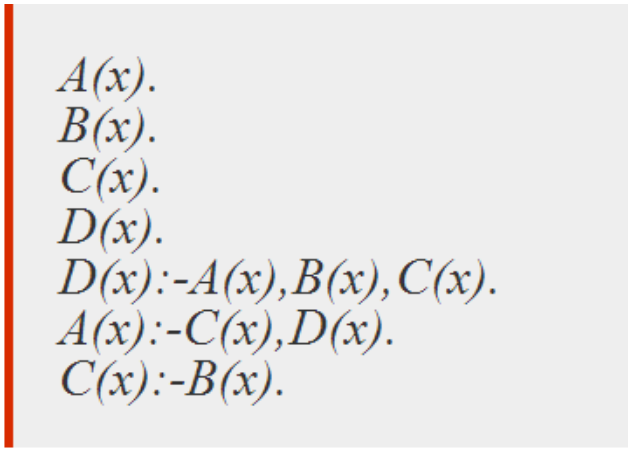
Nyní bude zobrazen výstup pro zadání formulí, které bylo zobrazeno na obrázku 8. Zadanými formulemi byly:

A(x)								
B(x)								
C(x)								
D(x)								
[[A(x)	&	B(x)]	&	C(x)]	>	D(x)		
[! C(x)	|	! D(x)]	|	A(x)				
! [B(x)	&	! C(x)]						

Obrázek 10: Zadání vstupních dat ve webové aplikaci

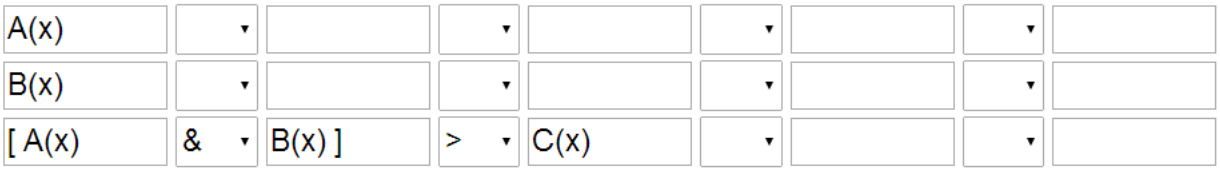
Nejdříve aplikace vyhodnotí ty formule, které v logickém programu budou představovat fakta. Jakmile budou zpracována všechna fakta, tak je spuštěn cyklus, který zpracuje formule, které budou představovat pravidla logického programu.

Následně bude zobrazen logický program, který odpovídá syntaxi programovacího jazyka Prolog.



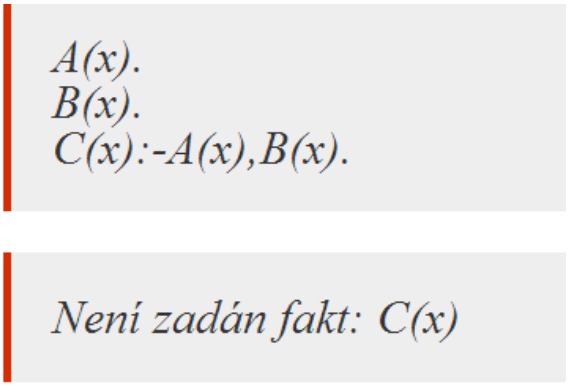
Obrázek 11: Logický program v syntaxi Prologu

V mé aplikaci je naimplementována i kontrola, zda veškeré fakty, které jsou zadány ve vstupních pravidlech jsou zároveň i v databázi programu. Pokud se v pravidlu vyskytuje fakt, který není uložen v databázi, tak je na tuto skutečnost uživatel upozorněn. Viz následující příklad:



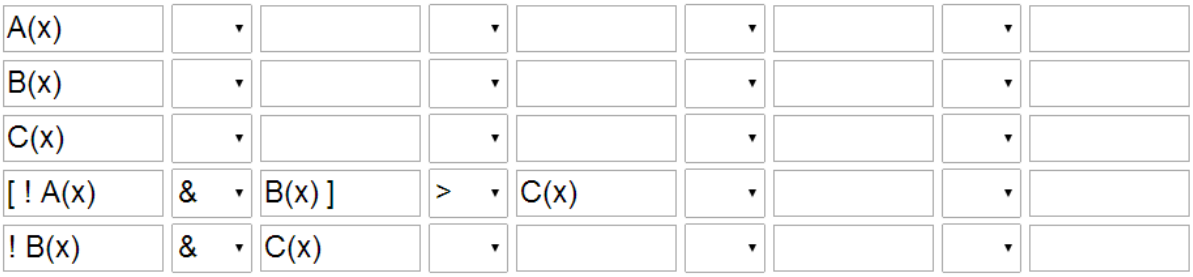
Obrázek 12: Zadání vstupních dat

Toto zadání vydá následující výstup:



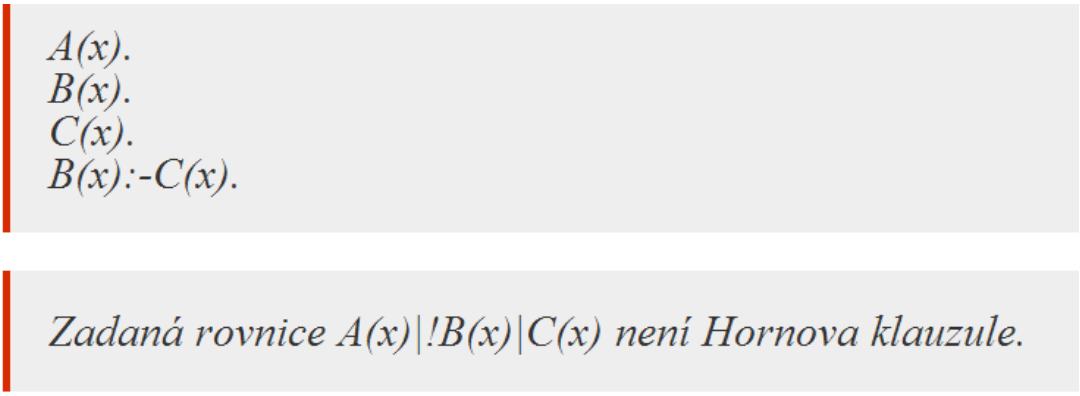
Obrázek 13: Výstup aplikace

Další možností, která může nastat je, že uživatel zadá rovnici, která není Hornovou klauzulí. Tudíž není možno s touto formulí v logickém programu Prologu pracovat. Logický program může přijmout pouze formule, které mají maximálně jeden pozitivní lirerál. Příklad, ve kterém tato skutečnost nastane je zobrazen níže:



Obrázek 14: Zadání vstupních dat

Toto zadání vydá následující výstup:



Obrázek 15: Výstup aplikace

5. Závěr

V této diplomové práci je v ucelené formě popsána teorie predikátové logiky prvního řádu. Je zde popsána syntaxe a sémantika jazyka a také důkazové metody. Hlavní zaměření je kladeno na obecnou rezoluční metodu.

Další kapitola je věnována logickému programování. Hlavní zaměření je kladeno na programovací jazyk Prolog. Jsou popsány rysy jazyka a strategie řízení výpočtů. Dále jsou zde popsány Hornovy klauzule a jejich omezení.

Hlavním cílem této diplomové práce byla aplikace řešící problematiku převodu formulí predikátové logiky 1. řádu na Hornovy klauzule. Byla vytvořena webová aplikace, která byla naprogramována na platformě ASP.NET. Logická vrstva aplikace je naprogramována v jazyce C#. Pro ukázku funkčnosti aplikace byl vytvořen demonstrační web.

Další vývoj aplikace je možný. Jednou z možností, které by bylo možno implementovat je načítání kompletního zadání ze souboru. A poté následné uložení logického programu do požadovaného formátu.

6. Literatura

[1] Duží, M. (2012): *Logika pro informatiky.* VŠB-TU Ostrava, Ostrava, ISBN: 978-80-248-2662-2.
[2] Cmorej, P. (2002): *Úvod do logické syntaxe a sémantiky*. Triton, Praha, ISBN:80-7254-294-X

[3] Raclavský, J. (2006): *Predikátová logika.* [Online] výukový text Masarykova univerzita Brno. http://www.phil.muni.cz/fil/logika/pl.php

[4] Sochor, A. (2001): *Klasická matematická logika.* Karolinum, Praha, ISBN: 80-246-0218-0

[5] Polák, J. (1992): *Prolog.* Grada, Praha, ISBN: 80-85424-36-3

[6] Jirků, P. (1991): *Programování v jazyku Prolog.* SNTL, Praha, ISBN: 80-03-00609-0

[7] Bratko, I. (2001): *PROLOG, Programming for Artificial Intelligence. Third edition.* Addison-Wesley, London, ISBN: 0-201-40375-7.

[8] Clocksin,W.F. - Mellish, C.S. (1987): *Programming in Prolog, Third, revised and Extended Edition.* Springer-Verlag, Berlin, ISBN: 3-540-17539-3.

[9] Degrout, D. – Lindstrom, G. (1986): *Logic programming-Functions, relations and equations.* Prentice-Hall, Englewood Cliffs, ISBN: 0-13-539958-0.